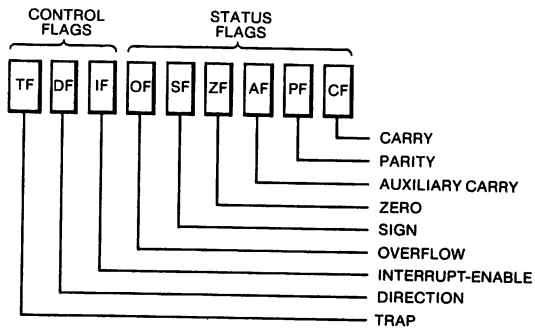
- Setting the trap flag (TF) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes each instruction.

Figure 7: Flags

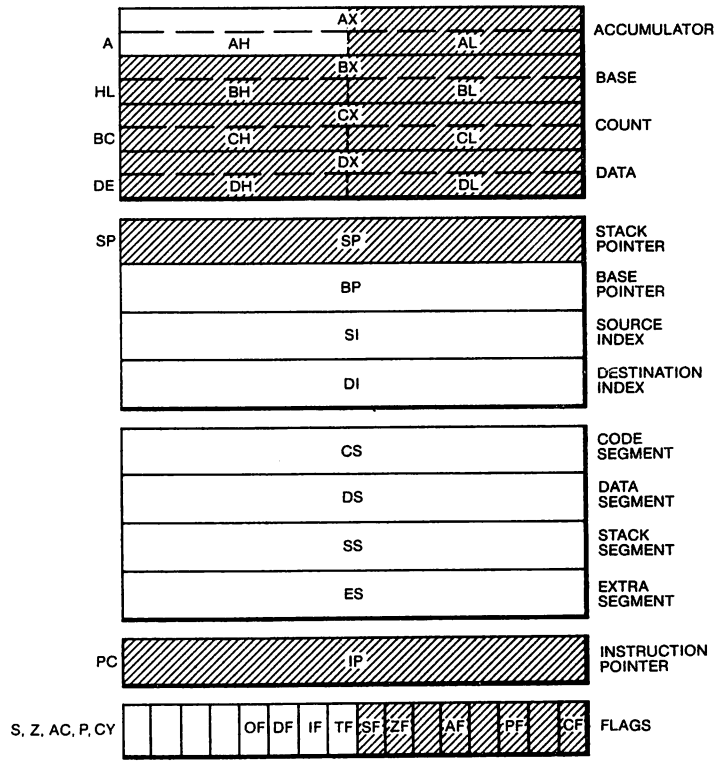


8080/8085 Register and Flag Correspondence

The registers, the flags, and the program counter in the 8080/8085 CPUs have counterparts in the 8088 CPU (see Figure 8). The A register (accumulator) in the 8080/8085 corresponds to the AL register in the 8088. The 8080/8085 H&L, B&C, and D&E registers correspond to registers BH, BL, CH, CL, DH, and DL, respectively, in the 8088. The 8080/8085 stack pointer (SP) and program counter (PC) correspond to the 8088 SP and IP.

The AF, CF, PF, SF, and ZF flags are the same in both CPU families. The remaining 8088 flags and registers are unique to the 8088. The 8080/8085 to 8088 mapping allows direct translation of most existing 8080/8085 program code into 8088 program code.

Figure 8: 8080/8085 Register Subset



Memory

The 8088 has 1,048,576 bytes of address space. This section describes how memory is functionally organized and used.

STORAGE ORGANIZATION The 8088 memory storage space is organized as an array of 8-bit bytes (see Figure 9). Instructions, byte data, and word data may be stored at any byte address, regardless of alignment. This technique saves storage space because code can be densely packed in memory (see Figure 10).

Figure 9: Storage Organization

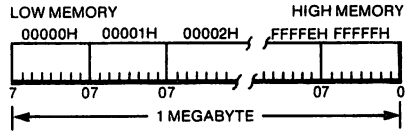
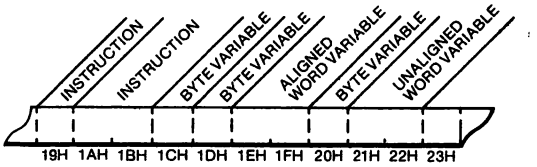
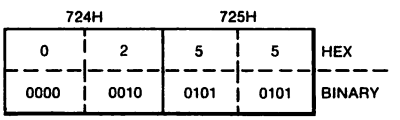


Figure 10: Instruction and Variable Storage



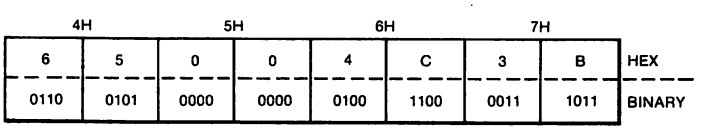
The most-significant byte in word data is always stored in the higher memory location (see Figure 11). This storage convention is "invisible" to the user except when the user monitors the system bus or reads memory dumps. A special class of data is stored as double words (i.e., two consecutive words) called pointers, which are used to address data and code outside the currently-addressable segments. The lower-addressed word of a pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally with the higher-addressed byte containing the most significant eight bits of the word (see Figure 12).

Figure 11: Storage of Word Variables



VALUE OF WORD STORED AT 724H: 5502H

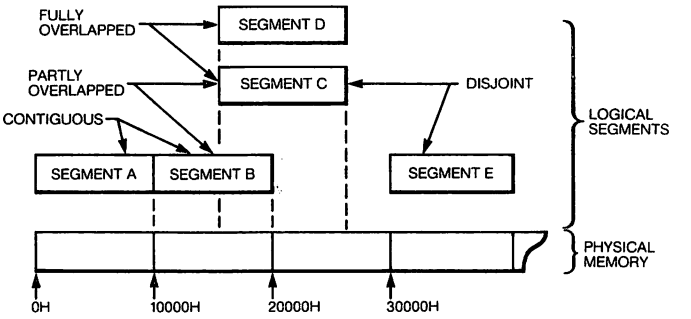
Figure 12: Storage of Pointer Variables



VALUE OF POINTER STORED AT 4H:
SEGMENT BASE ADDRESS: 3B4CH
OFFSET:65H

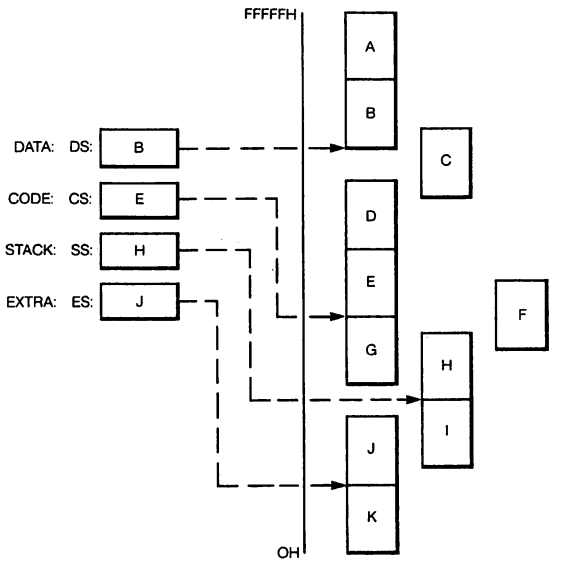
SEGMENTATION 8088 programs view the megabyte of memory space as a group of segments defined by the application. A segment is a logical unit of memory up to 64K bytes long. Each segment contains contiguous memory locations and is an independent, separately-addressable unit. Software assigns each segment a base address, which is the segment's starting location in the memory space. All segments begin on 16-byte memory boundaries. The segments can be disjoint, partially overlapped, or fully overlapped (see Figure 13). A physical memory location can be mapped into (contained in) one or more logical segments.

Figure 13: Segment Locations in Physical Memory



The segment registers contain (point to) the base address values of the four currently addressable segments (see Figure 14). Programs access code and data in other segments by changing the segment registers to point to the segments containing the needed code or data.

Figure 14: Currently Addressable Segments



Individual applications define and use segments differently. The currently-addressable segments provide a generous work space: 64K bytes for code, a 64K byte stack, and 128K bytes of data storage. Many applications can be written that simply initialize the segment registers and then forget them. However, large applications should be designed with careful consideration given to segment definition.

The segmented structure of the 8088 memory space supports modular software design and discourages the development of huge, monolithic programs.

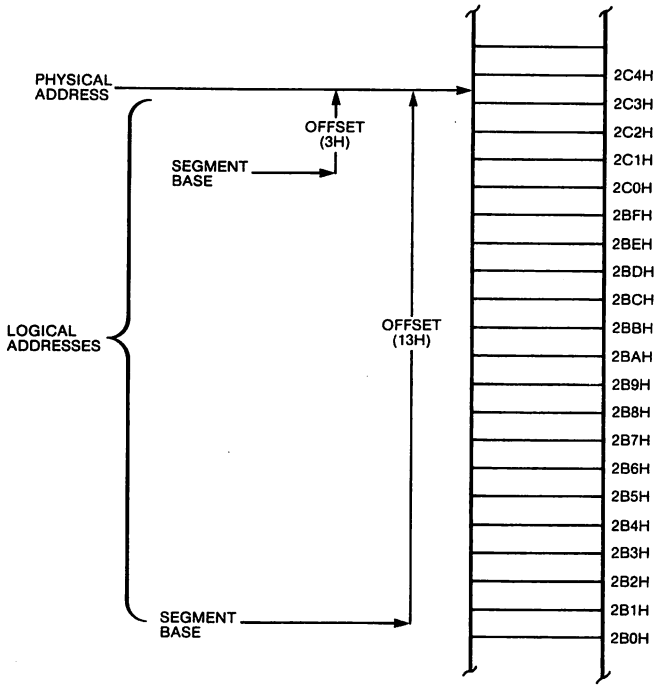
The segments can be used to advantage in many programming situations—for example, when programming an editor for several on-line terminals. A 64K text buffer (probably an extra segment) could be assigned to each terminal. A single program could maintain all the buffers by simply changing register ES to point to the buffer of the terminal requiring service.

PHYSICAL ADDRESS GENERATION There are two kinds of memory location addresses: physical and logical. A physical address is a 20-bit value that identifies each byte location in the megabyte memory space. Physical-address range varies from OH through FFFFFH. All exchanges between the CPU and memory components use physical addresses.

Programs use logical addresses, which allow code to be developed before the code is assigned physical addresses. This technique facilitates dynamic management of memory resources.

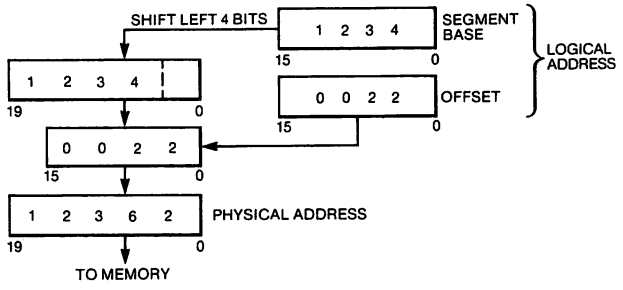
A logical address consists of two values: a segment-base value and an offset value. The segment-base value for any memory location is the value that defines the first byte of the segment. The offset value is the number of bytes from the beginning of the segment to the target location. Segment-base and offset values are unsigned 16-bit quantities. The lowest addressed byte in a segment has an offset value of 0. Different logical addresses can map to the same physical location, as shown in Figure 15. The physical memory location 2C3H shown in Figure 15 is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

Figure 15: Logical and Physical Addresses



When the BIU accesses memory to fetch an instruction, or to obtain or store a variable, it generates a physical address from a logical address. It does this by (1) shifting the segment-base value four bit positions, and (2) adding the offset value, as illustrated in Figure 16. This addition process results in modulo 64K addressing, which causes addresses to wrap around from the end of a segment to the beginning of the same segment.

Figure 16: Physical Address Generation



The BIU obtains the logical address of a memory location from different sources, depending on the type of reference that is being made (see Table 2). Instructions are always fetched from the current code segment. The IP contains the offset of the target instruction from the beginning of the segment. Stack instructions always operate on the current stack segment. The SP contains the offset of the top of the stack. Most memory operands reside in the current data segment, although the program can instruct the BIU to access a variable in one of the other currently addressable segments. The offset of a memory variable is calculated by the EU; the calculation is based on the addressing mode specified in the instruction, and the result is called the operand's effective address (EA).

Table 2: Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction fetch	CS	NONE	IP
Stack operation	SS	NONE	SP
Variable (except following)	DS	CS, ES, SS	Effective address
String source	DS	CS, ES, SS	SI
String destination	ES	NONE	DI
BP used as base register	SS	CS, DS, ES	Effective Address

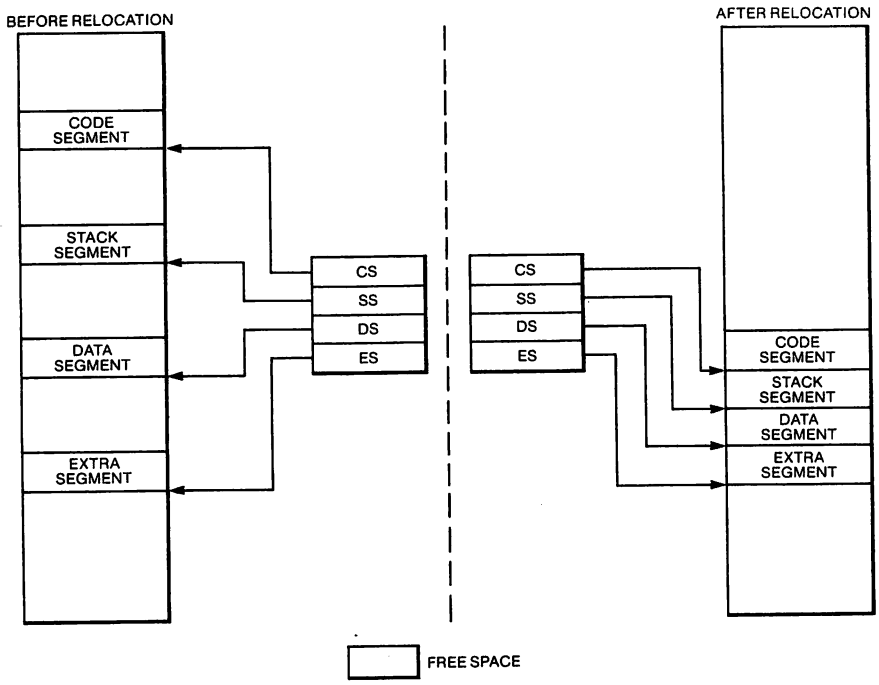
Strings are addressed differently than other variables. The source operand of a string instruction usually lies in the current data segment; however, another currently-addressable data segment may be specified. The source operand's offset is taken from register SI (the source index register). The destination operand of a string instruction always resides in the current extra segment, and its offset is taken from DI (the destination index register). The string instructions automatically adjust SI and DI as they process the strings one byte or word at a time.

When register BP (the base pointer register) is designated as a base register in an instruction, the variable is assumed to reside in the current stack segment. Using register BP is a convenient way to address data on the stack. The BP register can be used to access data in any of the other currently addressable segments.

Programmers usually find the segment assumptions of the BIU convenient to use. A programmer can, however, direct the BIU to access a variable in any of the currently-addressable segments by preceding an instruction with a segment override prefix. This 1-byte machine instruction tells the BIU which segment register to use to access a variable referenced in the following instructions. The only exception to this is a string instruction's destination operand, which must be located in the extra segment.

DYNAMICALLY RELOCATABLE CODE Dynamically relocatable—or position-independent—programming is made possible by the segmented memory structure of the 8088. The dynamic relocation technique makes effective use of available memory by taking advantage of the system's multiprogramming/multitasking capabilities. Inactive programs can be written to disk, making the space they occupied available to other programs. A disk-resident program can be read back into any available memory location and restarted. When a program needs a large contiguous block of storage and only nonadjacent fragments are available, other program segments can be compacted to free up a contiguous space (Figure 17).

Figure 17: Dynamic Code Relocation



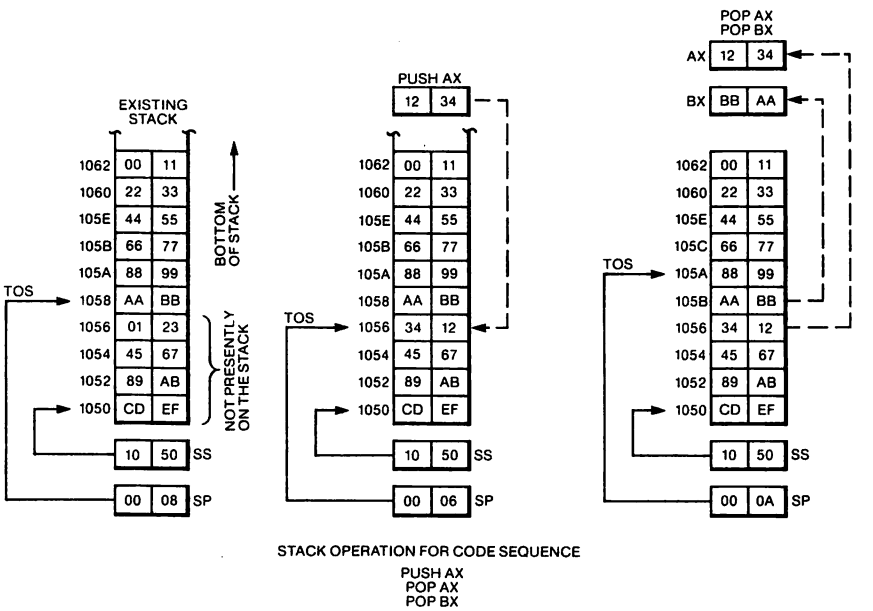
To be dynamically relocatable, all offsets in the program must be relative to fixed values contained in the segment registers. This allows the program to be moved anywhere in memory as long as the segment registers are updated to point to the new base addresses. A dynamically relocatable program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment.

STACK IMPLEMENTATION Stacks in the 8088 are implemented in memory. They are located by the SS (the stack segment register) and the SP (the stack pointer register). A system may have an unlimited number of stacks. Each may be the maximum length of a segment, 64K bytes.

Attempting to expand a stack beyond 64K bytes overwrites the beginning of the stack. Only one stack is directly addressable at a time; this stack is the current stack, often referred to simply as "the" stack. SS contains the base address of the current stack. SP contains the offset of the top of the stack from the stack segment's base address. The stack's base address (contained in SS) is not the "bottom" of the stack.

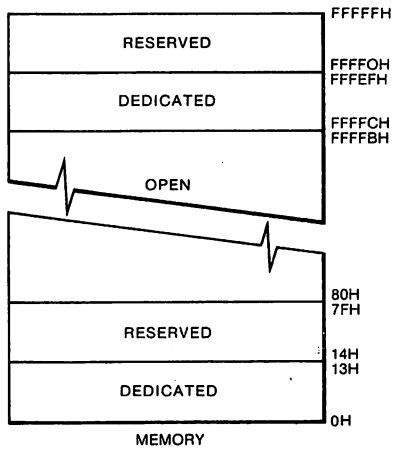
Stacks are 16 bits wide. Instructions that operate on a stack add and remove stack items one word at a time. An item is pushed onto the stack (see Figure 18) by decrementing SP by 2 and writing the item at the new TOS (top of stack). An item is popped off the stack by copying it from TOS then incrementing SP by 2. In other words, the stack grows down in memory toward its base address. Stack operations never move or erase items on the stack. The TOS changes only as a result of updating the stack pointer.

Figure 18: Stack Operation



DEDICATED AND RESERVED MEMORY LOCATIONS Two areas in extremely low and high memory—0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes)—are dedicated to specific processor functions or are reserved for use by hardware and software products (Figure 19). These areas are reserved for interrupt and system reset processing, and should not be used for any other purpose.

FIGURE 19: Reserved and Dedicated Memory



8086/8088 MEMORY ACCESS The 8088 always accesses memory in bytes. Word operands are accessed in two bus cycles, regardless of their alignment. Instructions are also fetched one byte at a time. Although word operand alignment does not affect performance, locating 16-bit data on even addresses ensures maximum throughput if the system is transferred to an 8086.

Input/Output

MEMORY-MAPPED I/O I/O devices may be placed in the 8088 memory space. The CPU cannot tell the difference between I/O devices as long as each device responds as a memory component.

Memory-mapped I/O provides programming flexibility. Instructions that normally reference memory may be used to access an I/O port located in the memory space. The move (MOV) instruction, for example, can transfer data between any 8088 register and a port. AND, OR, and TEST instructions may be used to manipulate bits in I/O device registers. Memory-mapped I/O takes advantage of the 8088 memory addressing modes. For example, a group of terminals can be treated as an array in memory with an index register selecting a terminal in the array.

However, a price is paid for the added programming flexibility that memory-mapped I/O provides. Dedicating part of the memory space to I/O devices reduces the number of addresses available for memory (although with a megabyte of memory space this should rarely be a constraint). Also, memory reference instructions take longer to execute and are less compact than simpler IN and OUT instructions.

DIRECT MEMORY ACCESS The 8088 provides hold (HOLD) and hold acknowledge (HLDA) signals that are compatible with traditional DMA controllers. By activating HOLD, a DMA controller can request use of the bus for direct transfer of data between an I/O device and memory. The CPU responds by completing the current bus cycle (if one is in progress) and then issuing HLDA, which grants the bus to the DMA controller. The CPU does not attempt to use the bus until HOLD goes inactive.

WAIT AND TEST The 8088 can be synchronized to an external event with the WAIT (wait for TEST) instruction and the TEST input signal. When the EU executes a WAIT instruction, the result depends on the state of the TEST input line. If TEST is not connected to or receiving an external signal, the processor enters an idle state and repeatedly retests the TEST line at 5-clock intervals. If TEST is connected to an external signal source, execution continues with the instruction following the WAIT.

The TEST input is connected to a "byte ready" signal from the floppy disk controller. This allows the processor to synchronize data transfer operations.

Processor Control And Monitoring — Interrupts

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors, and other components receive efficient servicing to ensure that the microcomputer can perform a large number of system tasks with little or no effect on throughput.

One desirable method for ensuring efficient servicing is to allow the microprocessor to execute its main program, stopping to service peripheral devices only when told to do so by the device itself. In effect, this method provides an external asynchronous input which informs the processor to complete whatever instruction is currently being executed and to fetch a new routine to service the requesting device. Once this servicing is complete, the processor resumes exactly where it left off.

The 8088 interrupt system is a simple and versatile interrupt system. Every interrupt is assigned a type code that identifies it to the CPU. The 8088 can handle up to 256 different interrupt types. Interrupts may be initiated by devices external to the CPU, or they may be triggered by software interrupt instructions and, under certain conditions, by the CPU itself, as illustrated in Figure 20. Figure 21 illustrates the basic response of the 8088 to an interrupt. The next sections elaborate on the information presented in Figure 21.

Figure 20: Interrupt Sources

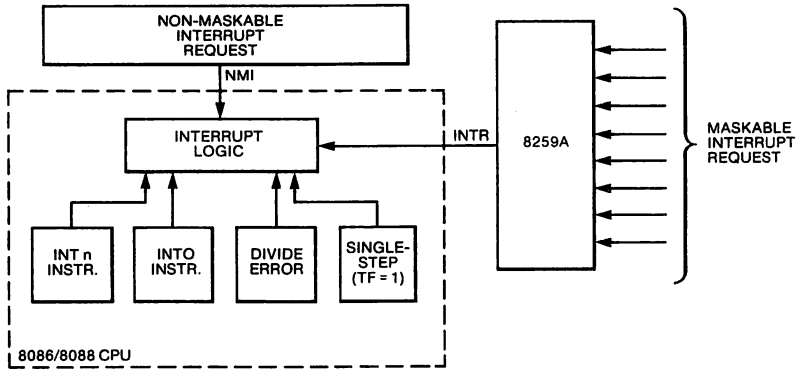
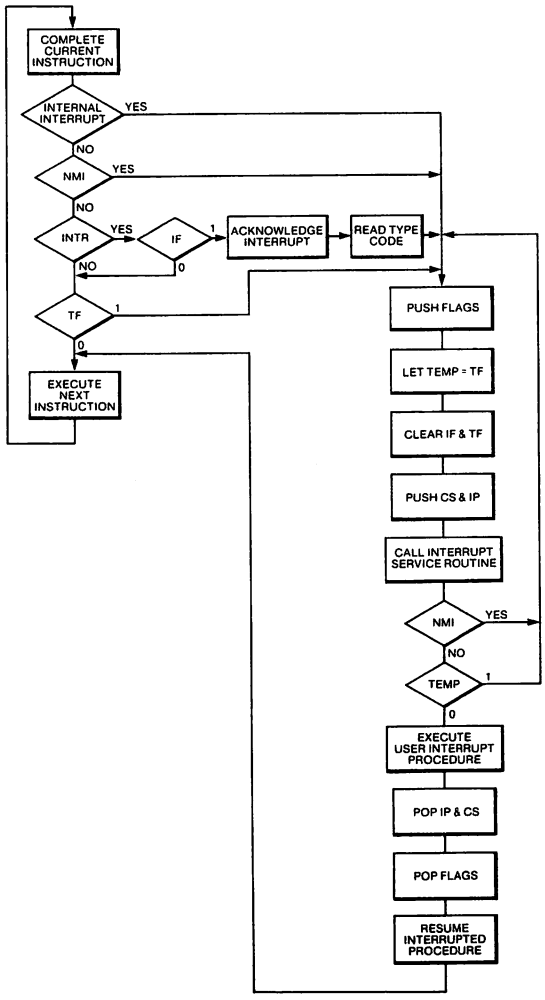


Figure 21: Interrupt Processing Sequence



EXTERNAL INTERRUPTS External devices can use two lines in the 8088 to signal interrrupts: interrupt request (INTR) and nonmaskable interrupt (NMI). The INTR line is driven by an 8259A programmable interrupt controller (PIC). The PIC is a flexible circuit controlled by software commands from the 8088.

The PIC appears as a set of I/O ports to the software and connects to devices that need interrupt services. It accepts interrupt requests from the attached devices and determines which service request has the highest priority. If the device selected for service has a higher priority than the one currently being serviced, the PIC activates the 8088 INTR line.

The CPU response to the active INTR line is based on the state of the interrupt-enable flag (IF). The currently-executing instruction is completed before the interrupt becomes active.

Occasionally, an interrupt request is not recognized until after the following instruction. Repeat, LOCK, and segment override prefixes are considered part of the instructions they prefix. Therefore, no interrupt is recognized between execution of a prefix and an instruction.

A move (MOV) to a segment register instruction and a POP segment register instruction are treated similarly (no interrupt is recognized until after the following instruction). This mechanism protects a program that is changing to a new stack (by updating SS and SP). The processor pushes the CS and IP flags into the wrong area of memory if an interrupt is recognized after SS has been changed, but before SP has been altered.

If a segment register and another value must be updated together, first the segment register must be changed, and then the instruction changing the other value must be given.

An interrupt request is recognized in the middle of an instruction in two instances—WAIT and repeated string instructions. In these cases, interrupts are accepted after any completed primitive operation or wait test cycles.

IF is clear when the interrupts signaled on INTR are masked or disabled, in which case the CPU ignores the interrupt request and processes the next instruction. The INTR signal is not latched by the CPU. It must be held active until a response is received or the request is withdrawn. When IF is set—enabling interrupts on INTR—the CPU recognizes the interrupt request and processes it. Interrupt requests arriving on INTR are enabled by executing a set interrupt-enable flag (STI) instruction, and disabled by executing a clear interrupt-enable flag (CLI) instruction. Writing commands to the 8259A (the PIC chip) selectively masks some of these requests. STI and IRET instructions re-enable interrupts only after the end of the following instruction, which reduces excessive stack buildup.

The CPU acknowledges an interrupt request by executing two consecutive interrupt acknowledge (INTA) bus cycles. Bus hold requests are not honored until INTA cycles are completed. The first INTA cycle signals to the 8259A that the request has been honored. The 8259A responds during the second INTA cycle by placing the interrupt byte containing the interrupt type (0-255) associated with the requesting device on the data bus. (Type assignment is made when the 8259A is initialized by software in the 8088.) The CPU uses this type code to call the indicated interrupt procedure.

A nonmaskable interrupt (NMI) request can arrive on another CPU line from an external source. This edge-triggered line signals to the CPU that a catastrophic event—such as the imminent loss of power, a memory error detection, or a bus parity error—has occurred. Interrupt requests arriving on NMI cannot be disabled. They are latched by the CPU and have a higher priority than an interrupt requested on INTR (level-triggered). NMI is first recognized when an interrupt request arrives on both lines during execution of an instruction. Nonmaskable interrupts are predefined as type 2. The processor does not need a type code to call the NMI procedure and does not run the INTA bus cycles in response to an NMI request.

The time required for the CPU to recognize an external request is determined by the number of clock cycles remaining to complete the instruction currently being executed. This delay is referred to as interrupt latency. The longest possible interrupt latency occurs when an interrupt request arrives during multiplication, divison, variable-bit shift, or rotate instruction execution. In the most extreme case, interrupt latency spans two instructions, rather than one.

INTERNAL INTERRUPTS Execution of an interrupt (INT) instruction generates an immediate interrupt. The interrupt type code identifies the procedure needed to process the interrupt. Since any type code can be specified, software interrupts can be used to test interrupt procedures that are written to service external devices.

When the overflow flag (OF) is set, an interrupt on overflow (INTO) instruction (a type 4 interrupt) is initiated immediately after the completion of the currently executing instruction. The CPU generates a type 0 interrupt following execution of a divide (DIV) instruction or an integer divide (IDIV) instruction when the calculated quotient is larger than the specified destination. When the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt after every instruction. This single-step execution, which is a powerful debugging tool, is discussed in more detail later.

All internal interrupts (INT, INTO, divide-error, and single step) share these characteristics:

- The interrupt type code is contained in the instruction or is predefined.
- No INTA bus cycles are run.

- Except for single-step interrupts, internal interrupts cannot be disabled.

- Internal interrupts (except single-step) have higher external interrupts (see Table 3). When interrupt requests arrive on NMI and/or INTR during execution of an instruction that causes an internal interrupt (e.g., a divide error), the internal interrupt is processed first.

Table 3: Interrupt Priorities

INTERRUPT	PRIORITY
Divide error, INT n, INTO	Highest
NMI	
INTR	
Single-step	Lowest

INTERRUPT POINTER TABLE The interrupt pointer (or interrupt vector) table links an interrupt type code and its associated service procedure. The interrupt pointer table occupies the first 1K bytes of low memory. There may be up to 256 entries in the table, one for each interrupt type that can occur in the system. Each entry in the table is a double-word pointer containing the address of the procedure servicing interrupts of that type. The higher-addressed word of the segment contains the procedure. The lower-addressed word contains the procedure's offset from the beginning of the segment. Each entry is four bytes long; the CPU calculates the location of the correct entry for a given interrupt type by simply multiplying the type number by 4.

In applications that do not recognize interrupt types, space at the high end of the table can be used for other purposes.

The 8088 activates an interrupt procedure by executing the equivalent of an intersegment indirect CALL instruction after pushing the flags onto the stack. The address contained in the interrupt pointer table element located at n × 4 (where "n" represents the type number) is the target of the CALL. The CPU saves the address of the next instruction by pushing CS and IP onto the stack. It transfers control to the interrupt procedure by replacing the second and first words of the table element.

The processor activates the interrupt procedures in priority order when multiple interrupt requests arrive simultaneously. Figure 22 shows how procedures would be activated in an extreme case. The processor is running in single-step mode with external interrupts enabled. INTR is activated during execution of a divide instruction. The instruction generates a divide error interrupt. Except for INTR, the interrupts are recognized in the order of their priorities (see Figure 23). INTR is not recognized until after the following instruction because recognition of the earlier interrupts cleared IF. If an earlier response to INTR is desired, interrupts can be re-enabled in any of the interrupt response routines.

All main-line code is executed in single-step mode (Figure 22). The processing speed (full speed or single-step mode speed) can be selected in each occurrence of the single-step routine because of the order of interrupt processing.

Figure 22: Processing Simultaneous Interrupts

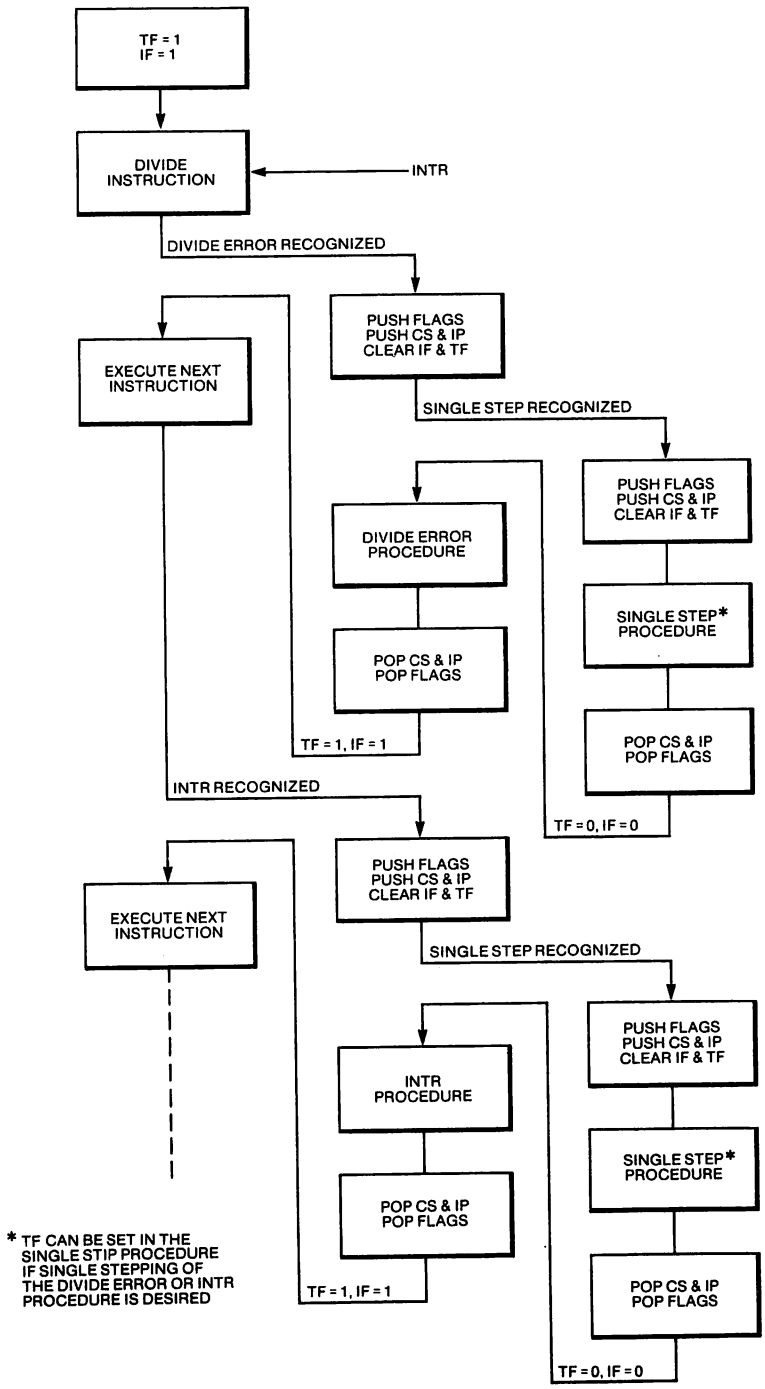
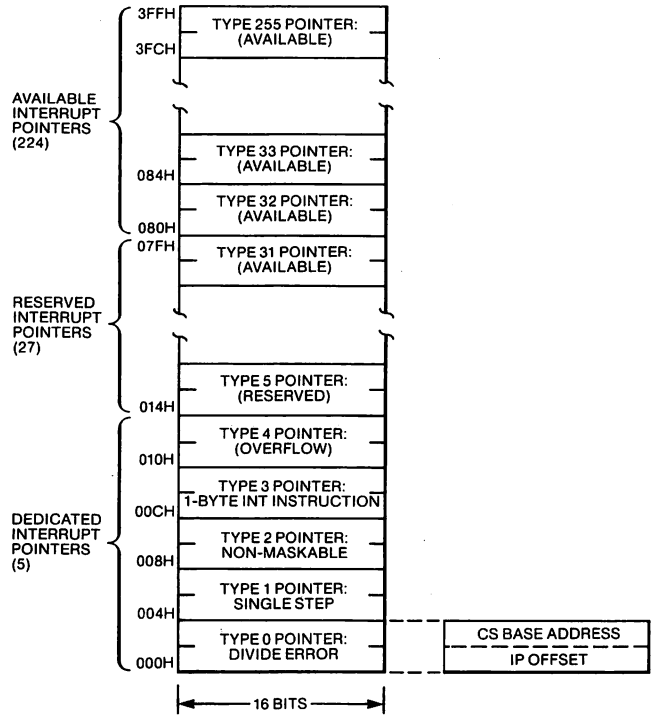


Figure 23: Interrupt Pointer Table



INTERRUPT PROCEDURES Flags CS and IP are pushed onto the stack and flags TF and IF are cleared when an interrupt service procedure is entered. The procedure can re-enable external interrupts with the set-interrupt-enable flag (STI) instruction, allowing itself to be interrupted by a request on INTR. Interrupts are not actually enabled until the instruction following STI has executed. An interrupt procedure can always be interrupted by a request arriving on NMI. The interrupt procedure can also be interrupted by software- or processor-initiated interrupts occuring within the procedure. (Programmers should ensure that the type of interrupt being serviced does not inadvertently occur during the interrupt procedure. For example, attempting to divide by 0 in the divide error (type 0) interrupt procedure results in the procedure being reentered endlessly.) Sufficient stack space must be available to accommodate the maximum depth of interrupt nesting that occurs in the system.

Prior to procedure termination, any registers used by the interrupt procedures should be saved before they are updated and restored. External interrupts for all sections except those sections of code that cannot be interrupted without risking erroneous results should be enabled. Interrupt requests on INTR can be lost if external interrupts are disabled for too long in a procedure.

Interrupt procedures with an interrupt return (IRET) instruction should be terminated. The IRET instruction assumes that the stack is in the same condition as when the procedure was entered. It pops the top three stack words into IP, CS, and the flags, and returns to the instruction that was to be executed when the interrupt procedure was activated.

The actual processing done by the procedure is application dependent. When servicing an external device, the procedure sends a command to the device, instructing it to remove its interrupt request. It can then read status information from the device, determine the cause of the interrupt, and act accordingly.

A software-initiated interrupt procedure can be used as a service routine (supervisor call) for other programs in the system. In this case, the procedure is activated when a program, rather than an external device, needs attention. (The "attention" might be to search a file for a record, send a message to another program, request an allocation of free memory, etc.) Software interrupt procedures can be used to advantage in systems that dynamically relocate programs during execution. Since the interrupt pointer table is at a fixed storage location, procedures can call each other through the table by issuing software interrupt instructions. This provides a stable communication exchange, independent of procedure addresses. Interrupt procedures can be moved if the interrupt pointer table is always updated, providing linkage from the calling program via the interrupt type code.

The 8088 is in single-step mode when the trap flag (TF) is set. In this mode, the processor automatically generates type 1 interrupt processing. The CPU automatically pushes the flags onto the stack and then clears TF and IF. The processor is not in single-step mode when the single-step interrupt procedure is entered. The old flag image is restored from the stack when the single-step procedure terminates, placing the CPU back into single-step mode.

Single stepping is a valuable debugging tool. A single-step procedure acts as a window into the system, through which operations can be observed on an instruction-by-instruction basis. A single-step interrupt procedure prints or displays register contents, instruction pointer values, key memory variables, etc., as they change after each instruction. This permits the exact flow of a program to be traced in detail. The point at which discrepancies occur can be identified by a single-step routine. A single-step routine can be used to accomplish the following:

- Writing a message when a specified memory location or I/O port changes value (or equals a specified value)
- Providing diagnostics selectively (for instance, only for certain instruction addresses)
- Letting a routine execute a number of times before providing diagnostics

The 8088 does not have instructions for setting or clearing TF. TF can be changed by modifying the flag image on the stack. The PUSHF and POPF instructions push and pop the flags. (TF can be set by ORing the flag image with 0100H. Clear TF by ANDing it with FEFFH.) After TF is set, the first single-step interrupt occurs after the first instruction following the IRET from the single-step procedure has been executed.

If the processor is single stepping, it processes an interrupt (either internal or external) as follows:

1. Control is passed normally (flags, CS and IP are pushed) to the procedure designated for handling the type of interrupt that has occurred.

2. Before the first instruction of that procedure is executed, the single-step interrupt is recognized and control is passed normally (flags, CS and IP are pushed) to the type 1 interrupt procedure.

3. When single-step procedure terminates, control returns to the previous interrupt procedure. Figure 23 illustrates this process in a case where two interrupts occur when the processor is in single-step mode.

BREAKPOINT INTERRUPT A type 3 interrupt is a breakpoint interrupt. A breakpoint is any place in a program where normal execution is arrested so that some sort of special processing may be performed. Breakpoints are inserted into programs during debugging to display registers, memory locations, etc., at crucial points in the program.

The INT 3 (breakpoint) instruction is one byte long, which facilitates planting a breakpoint anywhere in a program. The processor can be placed in single-step mode by using a breakpoint procedure.

Breakpoint instructions can insert new instructions (patch) into a program without recompiling or reassembling it. This can be done by saving an instruction byte and replacing it with an INT 3 (CCH) machine instruction. The breakpoint procedure contains new machine instructions—code to restore the saved instruction byte and decrement IP on the stack before returning control to the program. The displaced instruction is executed after the patch instructions.

NOTE: Undertake patching a program with caution. This action requires machine-instruction programming and can add new bugs to a program. Also note that a patch is only a temporary measure to be used in exeptional conditions. The affected code should be updated and retranslated as soon as possible.

SYSTEM RESET The 8088 RESET line provides an orderly way to start or restart an executing system. When the processor detects the positive-going edge of a pulse on RESET, it terminates all activities until the signal goes low, at which time it initializes the system as shown in Table 4.

Table 4: CPU State Following Reset

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Register	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

Since the code segment register contains FFFFH and the instruction pointer contains 0H, the processor executes its first instruction following system reset from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program. External (maskable) interrupts are disabled by system reset. As soon as the system is initialized, the system software should re-enable interrupts to the point where they can be processed.

PROCESSOR HALT When the halt (HLT) instruction is executed, the 8088 enters the halt state. This condition may be interpreted as "stop all operations until an external interrupt occurs or the system is reset." No signals are floated during the halt state, and the content of the address and data buses is undefined. A bus hold request arriving on the HOLD line is acknowledged normally while the processor is halted.

The halt state can be used when an event prevents the system from functioning correctly. An example might be a power-fail interrupt. After recognizing that loss of power is imminent, the CPU could use the remaining time to move registers, flags and vital variables to a battery-powered CMOS RAM area and then halt until the return of power was signaled by an interrupt or system reset.

Addressing Modes

The 8088 accesses instruction operands in many different ways. Operands can be in registers, instructions, memory, or I/O ports. Memory address and I/O port operands can be calculated several ways. These addressing modes extend the flexibility and convenience of the instruction set. This section briefly describes register and immediate operands, and then covers the 8088 memory and I/O addressing modes in detail.

REGISTER AND IMMEDIATE OPERANDS The quickest, most compact executing instructions specify only register operands. This is because register address is encoded in instructions in a very few bits, and the operation is performed entirely within the CPU (no bus cycles are run). Registers can be source operands and/or destination operands.

Immediate operands are constant data 8- or 16-bits long, contained in an instruction that is available directly from the instruction queue and can be accessed quickly. Like a register operand, no bus cycles are needed to obtain an immediate operand. Immediate operands are limited; they are constant values and can only serve as source operands.

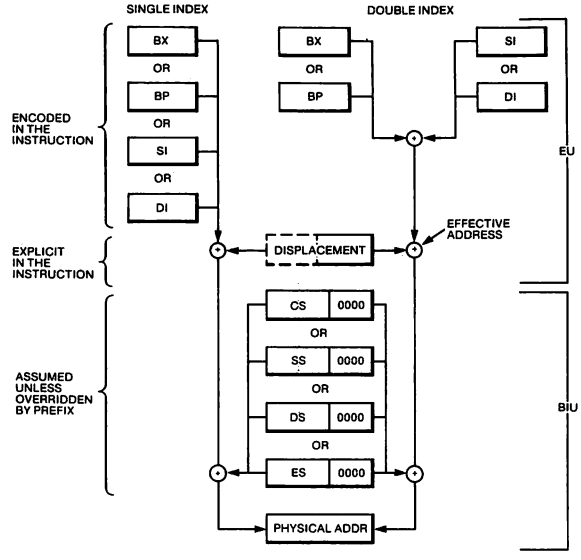
MEMORY ADDRESSING MODES Memory operands must be transferred to or from the CPU over the bus. The EU has direct access to register and immediate operands. When the EU needs to read or write a memory operand, it passes an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register, producing a 20-bit physical address. Then it executes the bus cycle(s) needed to access the operand.

EFFECTIVE ADDRESS The operand's effective address (EA) is the offset calculated by EU for a memory operand. EA is an unsigned 16-bit number expressing the operand's distance in bytes from the beginning of the segment in which it resides.

The EU calculates the EA in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the EA of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

Figure 24 shows that the execution unit calculates the EA by adding a displacement, the content of a base register, and the content of an index register. The variety of 8088 memory addressing modes results from combinations of these three components in a given instruction.

Figure 24: Memory Address Computation



The displacement, an 8- or 16-bit number contained in the instruction, is derived from the position of the operand name (a variable or label) in the program. A programmer can modify this value or specify the displacement.

A programmer can specify that BX or BP serve as a base register whose content is to be used in the EA computation. SI or DI can be specified as an index register. The displacement value can change the contents of the base and index registers can change during execution. This makes it possible for one instruction, as determined by current values in the base and/or index registers, to access different memory locations.

It takes time for EU to calculate a memory operand's EA. The more elements in the calculation, the longer it takes. Table 5 shows the time required to compute an effective address for any combination of displacement, base register, and index register.

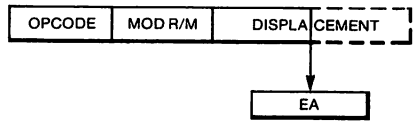
Table 5: Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP+DI, BX+SI	7
Displacement + Base + Index	BP+SI, BX+DI	8
	BP+DI+DISP	
Displacement + Base + Index	BX+SI+DISP	11
	BP+SI+DISP	
	BX+DI+DISP	12

*Add 2 clocks for segment override.

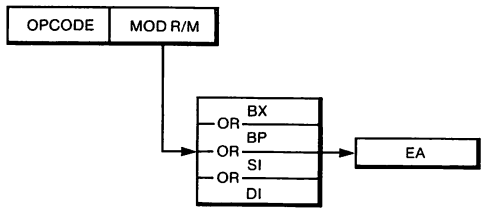
DIRECT ADDRESSING Direct addressing (see Figure 25) is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing is used to access simple variables (scalars).

Figure 25: Direct Addressing



REGISTER INDIRECT ADDRESSING The effective address of a memory operand can be taken from one of the base or index registers, as shown in Figure 26. When the value in the base of the index register is updated appropriately, one instruction can operate on many different memory locations. The load effective address (LEA) and arithmetic instructions change the register value.

Figure 26: Register Indirect Addressing



NOTE: Any 16-bit general register can be used for register indirect addressing with the JMP or CALL instructions.

BASED ADDRESSING In based addressing (Figure 27), the effective address is the sum of a displacement value and the content of register BX or register BP. Specifying BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). Therefore, based addressing with BP is a convenient way to access stack data.

Based addressing provides a straightforward way of addressing structures located at different places in memory (see Figure 28). A base register can be pointed at the base of the structure, and elements of the structure can be addressed by their displacements from the base. Different copies of the same structure can be accessed by changing the base register.

Figure 27: Based Addressing

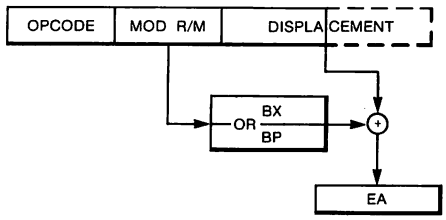
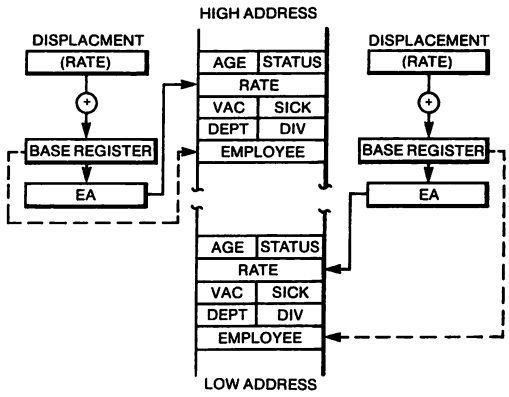


Figure 28: Accessing a Structure with Based Addressing



INDEXED ADDRESSING In indexed addressing, the effective address is calculated by the sum of a displacement plus the content of an index register (SI or DI) as shown in Figure 29. Indexed addressing is often used to access elements in an array (see Figure 30). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). All array elements are the same length, so simple arithmetic on the index register selects any element.

Figure 29: Indexed Addressing

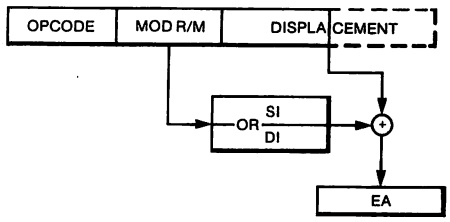
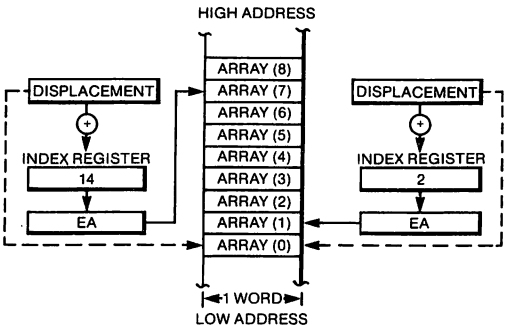


Figure 30: Accessing an Array with Indexed Addressing



BASED INDEXED ADDRESSING Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement (see Figure 31). Two address components can be varied at execution time, making based indexed addressing a very flexible mode. Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack (see Figure 32). Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

Based indexed addressing can access arrays contained in structures and matrices (two-dimension arrays).

Figure 31: Based Indexed Addressing

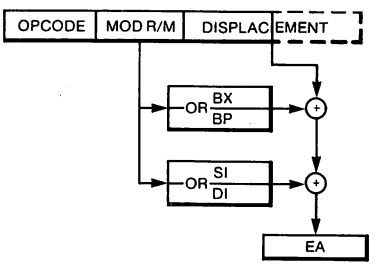
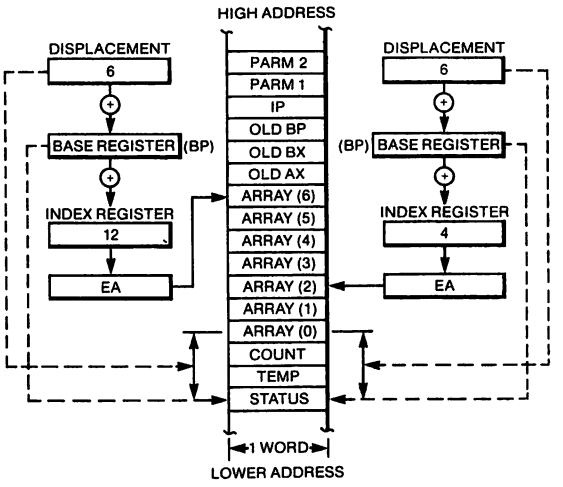
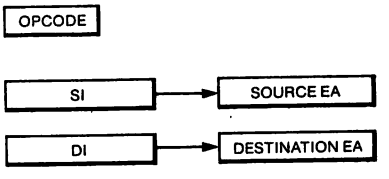


Figure 32: Addressing a Stack Array with Based Indexed Addressing



STRING ADDRESSING String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly as shown in Figure 33. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, the CPUs automatically adjust SI and DI to obtain subsequent bytes or words.

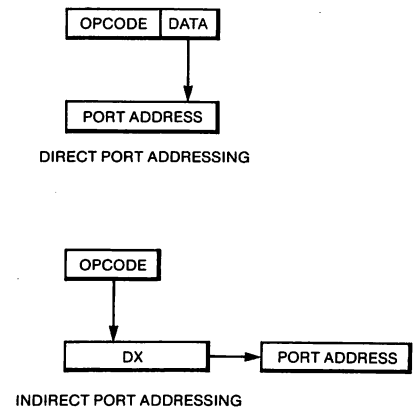
Figure 33: String Operand Addressing



I/O PORT ADDRESSING When an I/O port is memory mapped, any of the memory operand addressing modes can be used to access the port. For example, a group of terminals can be accessed as an array. String instructions can also transfer data to memory-mapped ports with an appropriate hardware interface.

The two addressing modes that can be used to access ports located in the I/O space are illustrated in Figure 34. In direct port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and ranges from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

Figure 34: I/O Port Addressing



Boot ROM

The boot ROM has up to 16K of memory. When the 8088 is reset or powered on, the microprocessor goes to the highest memory area and begins to execute code in the boot ROM. The boot ROM performs basic initialization of all hardware in the machine. It then tries to read the boot software in the disk drives, which contains the operating system. The boot software is loaded into the processor's system random access memory (RAM). When this process is completed, the boot ROM jumps into the operating system and begins executing in the operating system.

INPUT/OUTPUT (I/O) FUNCTIONS

The I/O function consists of serial ports, a parallel port, a control port, an audio input/output function, and a keyboard port.

Serial Ports

The standard configuration includes two full-duplex, serial communications ports. The serial ports are independent and are controlled by a single chip, the NEC 7201. These ports support the RS-232 standard serial interface and can be programmed for asynchronous and for more advanced protocols (e.g., SDLC and IBM binary synchronous communications). Each port is capable of running with an internally generated bit clock (or clocks) supplied by an external source (usually the MODEM). The clock selection is made under software control. There is a programmable bit clock generator for each channel to provide clocking if the internal mode is selected (channels 0 and 1 of the 8253 timer chip are used for this purpose).

Parallel Port

The parallel port is a dual function port supporting parallel Centronics and IEEE 488 interfaces. It is software configurable so as to support these interfaces. The Centronics interface is an 8-bit parallel output interface to standard printers and other devices; the IEEE 488 interface is an instrumentation interface. Initially developed by Hewlett-Packard, the IEEE 488 interface allows for multiple independent devices and for better control and more advanced functions than does the Centronics port. The parallel port is buffered with the standard IEEE 488 drivers.

Control Port

The control port is a series of stake pins on the main logic assembly that contain I/O lines from a 6522 I/O chip. There are two complete 8-bit I/O control ports. Each pin can be configured for input or output (to drive one standard TTL load).

Each 8-bit port has two handshake control lines. The only pin on the control port dedicated to another function is the most significant bit (MSB) of port B. This pin is dedicated to the audio clock that controls the sample rate for the audio. When the Codec audio is in use, the MSB is active.

The control port also has a light pen connection which connects to the CRT controller chip and to +12V, -12V, +5V, and ground signals. It supplies minimum power to an external device.

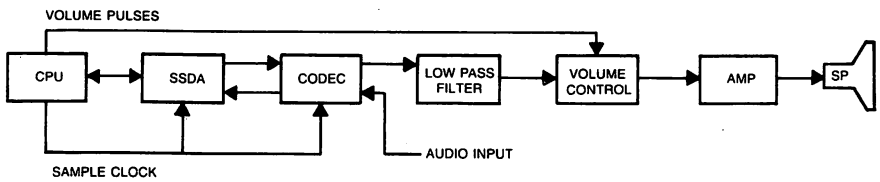
Audio Section

The audio section can generate voice, tones, bells, or other sounds through the speaker in the processor unit. The sounds are stored in a specially coded digitized form in the computer memory. The volume level of sounds generated by the processor unit can be controlled through software or directly with special keys on the keyboard. With additional hardware, the audio section also supports input from external analog sources, allowing digital recording of sounds for future playback.

As shown in Figure 35, the sound output function acts basically as a pipeline from the CPU to the speaker. Sound in digital byte form is stored in the CPU memory. The CPU transfers the sound bytes to the synchronous serial data adapter (SSDA). The SSDA converts the bytes into a serial bit stream of data to feed to the coder/decoder (Codec). The Codec converts the serial data into a varying analog signal. The analog signal is sent through a low pass filter to remove any high frequency noise generated in the digital-to-analog conversion in the Codec. The filtered analog signal is sent into a volume-control section. The volume-control section switches the

analog signal at a variable on-to-off rate, allowing the sound level to be controlled. The analog signal is finally sent through an audio amplifier to the speaker in the processor unit.

Figure 35: Audio Section Block Diagram



The synchronous serial data adapter (SSDA) is the major interface between the CPU and the audio section. The main function of the SSDA in playback mode is the buffering and conversion of 8-bit bytes into a serial bit stream for the Codec. In the record mode, the SSDA also converts a serial bit stream from the Codec into bytes for the CPU.

The SSDA is a 6852 I/O chip. The SSDA's control and data resisters are memory-mapped in the CPU's high memory space. The SSDA contains a 3-byte FIFO register buffer. The FIFO allows the CPU to fill the SSDA with three bytes of data and then perform other processing while the SSDA shifts bits out to the Codec. This reduces processor overhead while the processor is playing or recording sounds. The SSDA first shifts the data to the Codec's least significant bit. The SSDA control registers then tell the CPU that the FIFO is ready for more data. The SSDA also provides playback/record (decode/encode) control via its "DTR" output.

The CPU controls the sound quality of the audio section with the shift clock sent to the SSDA and the Codec. The shift clock is generated in one of the CPU's 6522 I/O chips. The PB7 output from the 6522 is controlled by an internal timer, which provides adjustable clock frequency. The higher the frequency of the shift clock, the better the sound quality. Because faster shift clocks require more memory to store the sound bytes, a trade off must be made between sound quality and memory storage. A shift clock of 16Khz will produce telephone quality reproduction of the original sound with each second requiring 2K bytes of storage.

The Codec converts digital data into analog signal in the playback mode and analog signal into digital data in the record mode. The Codec uses a technique known as delta modulation to convert the serial bit stream into analog output. The digital data's 0's and 1's are commands to the integrator in the Codec to make its analog output signal "go up" or "go down" respectively. The serial bit stream represents the direction for the analog output signal.

To increase dynamic range, continuously variable slope delta-modulation (CVSD) is used. An outstanding characteristic of CVSD is its ability, with fairly simple circuitry, to transmit intelligible voice sounds at relatively low data rates. CVSD increases the dynamic range by "companding" (compressing-expanding), which gives small signals a higher relative gain. The CVSD scheme detects three or more consecutive 0's or 1's in the data stream. When this occurs, the gain of the integrator is adjusted to ramp faster to track larger signals. Up to a limit, the more consecutive 1's or 0's, the larger the obtained ramp amplitude, and the better the reproduction of the original sound.

The low pass filter removes unwanted high frequency noise generated in the CODEC. The filter is set for a 3KHz cutoff frequency. This limits sounds to the normal voice bandwidth.

Volume is controlled by varying the duty factor of the analog signal from the filter. The CPU controls the volume level by switching the analog signal on and off at a frequency above the audible range. A minimum of 20KHz is recommended. The CPU uses a 6522's shift register in a recirculating output mode to generate the duty cycle for the volume control. This allows selection of seven different volume levels (and also off).

The final stage is a four watt audio power amplifer which drives the speaker mounted in the disk drive subassembly. A large speaker can be attached to produce more sound output.

Keyboard Interface

There are six signals, or lines, going to the keyboard from the processor. A +5V supply and a ground signal power the keyboard. A shield line shields the keyboard from static and interference. There are three signal lines: ready, data, and acknowledge.

The ready, data, and acknowledge lines control communications between the keyboard and the processor. The keyboard sends data to the microprocesser serially. The keyboard acknowledges or signals to the processor that a key signal has been received and is ready to be sent to the processor. It does this with a keyboard ready line. When the processor is ready, it handshakes the data in via the acknowledge line and the data comes across on the keyboard data line.

The keyboard uses the serial shift register capabilities of a 6522 interface chip to communicate with the microprocessor. This function is handled automatically by the 6522 until the whole key identifier has been received into the shift register. Then the processor reads the key identifier, and handshakes the final check bit sequence.

See Chapter 4, "Keyboard Unit," for a more detailed description of the keyboard interface.

DISK INTERFACE

The signals sent to the disk interface are 8-bit data lines, read/write signals, selection logic signals, and addressing and control signals. They control, send information to, and receive information from the disk drive assembly. A connector on the main logic assembly connects to the drive assembly through a cable. The main logic assembly and microprocessor control the drives with these signals while receiving and sending data to the drive assembly.

EXPANSION BUS

The main logic board supports expansion of the system through four female 50-pin edge connectors. These connectors provide an interface for memory expansion boards and special control boards. Some of the control boards are highspeed network systems, hard disk controller interfaces, and I/O expansion boards for use with science-related applications. The expansion bus has a set of data lines, addressing lines, control lines, and power lines capable of driving any expansion interface. Additional expansion capabilities provide external-device access to memory internal to the main logic assembly.

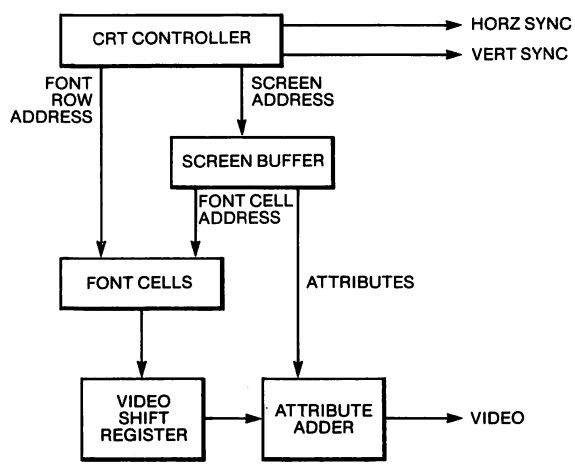
DISPLAY

Standard raster scanning techniques are used to display information on the screen. The most common mode of operation is the text mode, which displays 80 character cells horizontally by 25 lines vertically. This means that an electron beam, scanning horizontally, divides the screen into scan lines. The lines are scanned from left to right and top to bottom.

As the beam scans left to right, the CRT controller generates addresses for the screen buffer RAM. The CRT controller selects words from the screen buffer memory, determining the type of character and the attributes to be displayed. A character cell is 10 dots wide by 16 scan lines high in the text mode. These characters are RAM-mapped and programmable.

The lower 128K bytes of RAM (as well as the 4K bytes from F0000 to F0FFF) is dual port memory. One port of the lower 128K bytes of RAM is used by the display hardware to refresh the raster-scan display. The dual-port memory is managed by an arbitrator circuit that guarantees one refresh access to the display RAM every character cell time. The arbitrator circuit adds a wait-state to any 8088 memory cycle if this is necessary to isolate it from the display-refresh cycle. The display circuit manages the memory-refresh in the dual port on-board dynamic RAM.

Figure 36: Display System Block Diagram

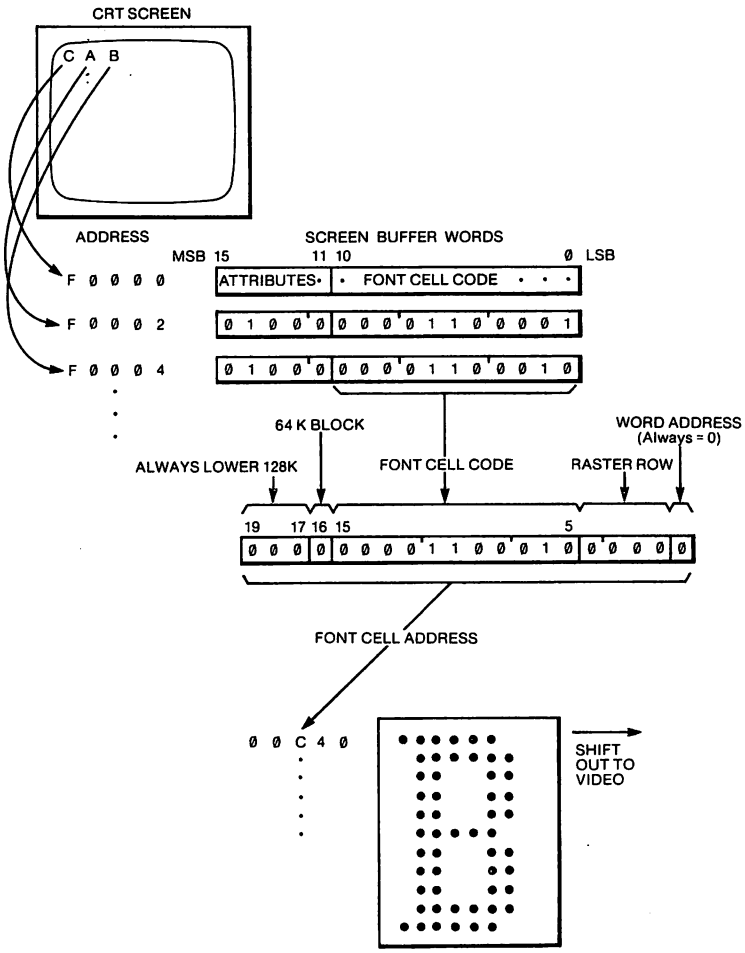


Screen Buffer

The screen buffer is a section of memory 2000 words in length (it is mapped at addresses F0000 through F0FFF).

The words are arranged linearly. The first word in the screen buffer defines the top leftmost character on the screen. The next word in the screen buffer defines the next character on the screen, reading left to right, and etc. All of the characters on the screen are defined in the screen buffer prior to display.

Figure 37: Display Operation



FONT POINTER The words in the screen buffer are broken into two pieces. The lower 11 bits comprise the font pointer. The upper five bits are attribute bits. The font pointer contains binary address information. Up to 2048 characters, or font cells, can be displayed on the screen.

ATTRIBUTE BITS There are five attribute codes associated with each character. Four of these attribute bits are used for reverse video, underline/strikeover, high/low-intensity, and nondisplay. The other bit is available for user software or external hardware.

Each character on the screen is affected by the attributes in the upper 5 bits. Each attribute bit is independent of the other bits.

Reverse Video The reverse video attribute displays black characters on a white background. This affects all the dots in every character, including underline and other modes.

Display High/Low Intensity The high/low intensity attribute displays a character in high intensity (enhanced mode), or in low intensity.

Display Underline/Strikeover The underline/strikeover attribute works in conjunction with the font cell control bit mentioned above. One bit in a font cell word determines where the underline/strikeover occurs (this is discussed later, in "Font Cell"). Underline creates a solid line through the character cell; thus, text underlining is programmable. It can also be used as a strikeover if the underline control bit is in the middle of the character rows. The strikeover is displayed on the screen and superimposed on the character when the attribute is turned on.

Nondisplay Attribute The nondisplay attribute suppresses dot information so that the character is not displayed on the screen.

Software Attribute The software is available for software application program use to identify special fields on the screen, mark the end of lines, or mark special text in an editor. It is not used for display generation functions.

The character and attribute bits are organized into words. The lower 11 bits of each word define which of the 2048 possible characters (font cells) is placed at that location on the screen. The upper five bits identify attributes. These words are on even address boundaries. The 80-character-by-25-line display occupies 2000 words (4000 bytes) of the screen memory.

The five attribute bits are sent to the video control section. The video control section adds the reverse video, intensity, cursor, underline, and nondisplay functions, according to the attribute bits.

The lower 11 bits are the font cell code. The font cell code has other address bits added to it—five lower bits and four upper bits—to generate a font cell address. The first four of the five lower bits, one through four, are the raster row. Using this binary code, 16 raster rows—the number of raster rows in a standard character—can be addressed.

The lower bit, bit 0, is the byte address bit. It is always a zero because words in memory for the font cell are being addressed.

The upper four bits select the 64K block of memory in which the font cells are located. The font cell RAM is limited to the lowest 128K of memory, so bit 17 through bit 19 are always zero.

When bit 16 is zero, it selects the lower 64K of memory. When bit 16 is one, it selects the next block of 64K of memory. This 15-bit address, bits 19 to 5, is the base of the font cell address. The display hardware then appends this address to the raster row being scanned. It takes the addressed word out of the font cell memory and passes it to the video shift register. The word is then processed through attribute control and out to the display.

Font Cell

Characters are generated using a high-density dot matrix technique resulting in a high-resolution display of characters on the screen. This technique uses a font cell as the basic structure within which characters are developed for display. The font cell is a sequential block of 16 words that are accessed to form a dot matrix 16 bits wide and 16 raster rows high.

The first word's least significant bit (LSB) is displayed at the top leftmost position of the font cell display. The second word's LSB is displayed at the leftmost position on the second line, and so forth, through all 16 scan lines. Ten dots of the 16-bit wide cell are displayed on each line. The remaining six dots of each word, which are most significant bits (MSBs), are not displayed.

The underline/strikeover control bit is the MSB of each font.

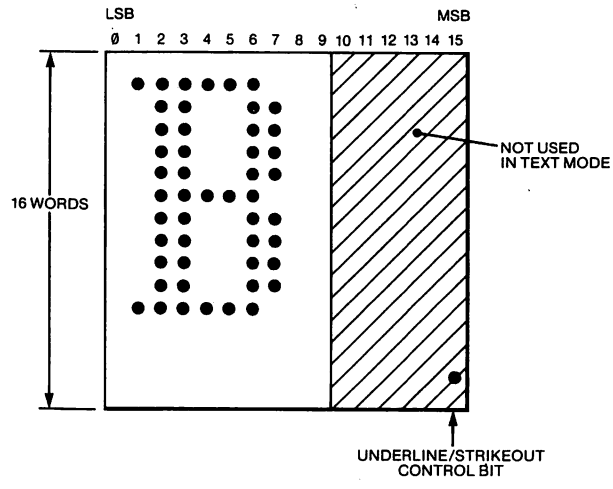
In normal mode, a bit value of 1 displays a white dot, and a bit value of 0 displays a black dot (in reverse video mode, the reverse is displayed). A word, which consists of 16 bits, defines the condition of each dot in the matrix (see Figure 38).

Figure 38: Font Cell Example

	LSB 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	MSB 15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	①	①	①	①	①	①	①	0	0	0	0	0	0	0	0
2	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
3	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
4	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
5	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
6	0	0	①	①	①	①	①	①	0	0	0	0	0	0	0	0
7	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
8	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
9	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
10	0	0	①	①	0	0	0	①	①	0	0	0	0	0	0	0
11	0	①	①	①	①	①	①	①	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	*
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

* = Underline/Strikeover Bit

Figure 39: Block Diagram of a Font Cell



To summarize, the CRT controller chip generates word addresses in the screen buffer memory. A portion of each word contains the attributes, which are passed to the video output section. Another portion of each word is the font cell code, which, when combined with other bits, generates a font cell address. The word at this font cell address is loaded into a video shift register which turns the parallel word into serial bits and passes it to the video output section, where it is combined with the attribute functions. The CRT controller chip also generates the horizontal/vertical signals that go to the display.

Display Brightness

Overall display brightness is software adjustable. Brightness may be adjusted to one of eight different levels by setting the brightness control bits (PB2, PB3, and PE4 of the 6522 at E8040) to the binary value corresponding to the desired level. The binary values range from zero to seven, in order of increasing brightness.

Display Contrast

Display contrast is also software adjustable. The contrast function controls the difference in intensity between high- and low-intensity characters. Only the intensity of the low-intensity characters is varied by the contrast function. Contrast may be adjusted to one of eight levels by setting the binary value of the desired level in the three contrast control bits (PB5, PB6, and PB7 of the 6522 at E8040). The binary values range from zero to seven, in order of increasing contrast (a binary value of zero causes no difference in contrast).

HIGH RESOLUTION MODE

A bit-mapped high-resolution mode is configured for 800 by 400 dots of bit-addressable display. In this mode, the reverse video, high/low-intensity, and nondisplay attributes apply to fixed 16- by 16-dot cells on the screen, and the underline/strikeover attribute is disabled.

The high-resolution mode makes special use of the font cell graphics. The output line (HIRES) controls the font cell width. When high, this line enables the 16-dot matrix, which displays all 16 bits of each font cell word. In this mode, the screen is organized into a 50-column by 25-line display.

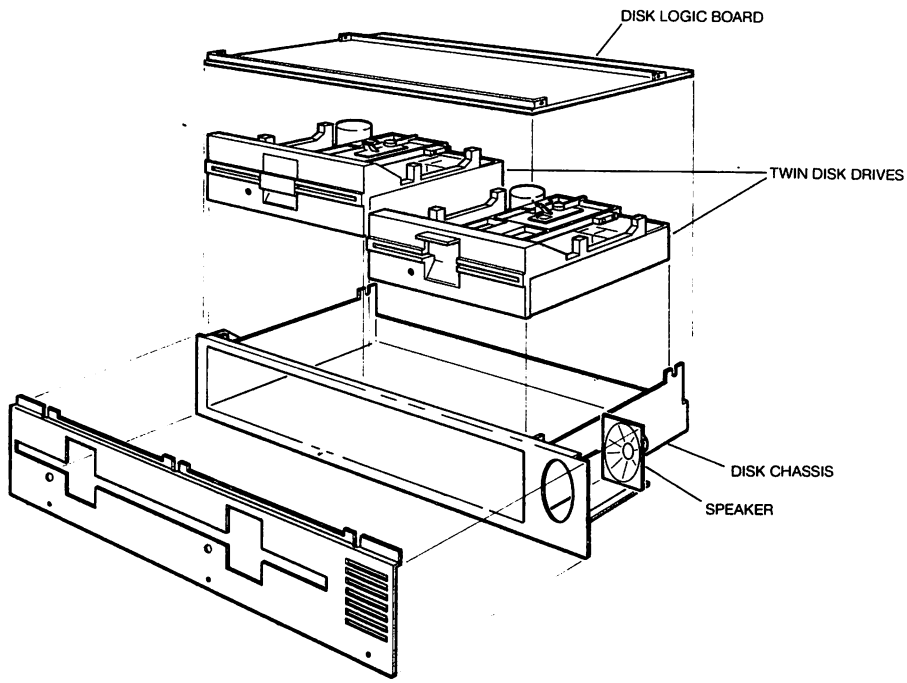
To use the bit-mapped display mode, the screen buffer is filled with font cell pointers which address successive font cells, by column. For example, if line 1/column 1 addresses font cell N, line 2/column 1 would address font cell N+1, and line 25/column 1 would address font cell N+24. Line 1/column 2 would address font cell N+25, and so forth. Line 25/column 50 which would address font cell N+1249. The font cell memory is directly manipulated, without further modification to the screen buffer.

In high-resolution mode, the programmer's view of the screen is 20,000 contiguous words of bit-mapped dots organized into 16-bit wide columns, going from top to bottom, and left to right as word addresses increase.

DISK DRIVE ASSEMBLY

As shown in Figure 40, the disk drive assembly is comprised of two floppy disk drive mechanisms, a disk drive interface board, and a chassis which also contains a speaker. The disk drive assembly provides the system with a minimum of 1.2 million bytes (formatted) of auxillary storage.

Figure 40: Disk Drive Assembly



The standard drive units are 5-1/4 inch, 80-track mechanisms, which operate with single-sided media. Track density is 96 tracks per inch, and recording density is maintained at approximately 8000 bits per inch on all tracks.

FUNCTIONAL DESCRIPTION

The disk drive interface board provides all the low level operations required to convert binary information for storage on and retrieval from diskette. Status and drive control interface to the drives is also provided on the disk drive interface board.

The processing unit maintains functional control of the disk drive assembly.

Reading Data

The 8088 CPU transfers data from the disk to memory as byte-by-byte read operations. Before the data is transferred, the drive motor for the drive containing the disk is started, and the head is positioned to the correct track. The GCR read circuit provides sync detection and separation. (Sync is a special GCR pattern that does not occur in normal data fields. The sync pattern consists of 10 ones during a byte time; other GCR patterns cannot contain more than 8 ones during a byte time.) When the GCR read circuit detects a sync mark, it starts a counter that causes an interrupt to be sent to the CPU, if sync remains present for 6 byte times. This interrupt to the CPU, which is called SYN and is on the highest level interrupt input line to the interrupt controller, informs the CPU that a header sync mark has been detected.

HEADER SEARCH When a sync interrupt occurs while the CPU is searching for a sector, the CPU enters the controller software that will compare the sector header information with the sector requested (the sector header contains the data block ID, track numbers, the sector number, and the checksum). This compare function is performed by the CPU on a byte-by-byte basis. The GCR read circuit provides a data byte every 21.3 microseconds. In order to be able to keep up with the high data rate, the CPU uses a special instruction (WAIT) that stops processing until a byte-ready strobe occurs on the test input. The CPU then continues processing by reading the latched data byte and comparing it with the requested sector information.

If the sector is not the correct sector, the CPU returns from the interrupt and continues processing until the next header sync interrupt. Once the desired sector header has been found, the data transfer can begin.

DATA TRANSFER Before the CPU can read the data block of a sector, the clock recovery circuitry must be resynchronized. This is required because the data block is updated and can be written at any random phase relative to the header information. The data block sync mark is only 5 bytes long and is not detected by the header sync mark detection circuit (header sync marks must be at least 6 bytes in length). The CPU polls the sync input line until the data block sync is detected and then verifies that the byte following sync—the data block IO byte—is correct. If it is not correct, a "not data block IO error" is generated, and no data is transferred. Using the WAIT

instruction, the CPU then transfers the following 512 bytes of sector information to the present destination in memory. As the CPU moves the data to memory, it also computes the checksum. This resulting checksum is then compared with the checksum recorded in the data block. If the checksums match, the data transfer is correct; otherwise, error recovery by the CPU is needed.

Writing Data

Data transfer from memory to disk is performed by the CPU in much the same manner as for read operations. The disk drive motor is started and set to the proper speed, and the head is positioned at the correct track by the controller software. The CPU does a header search using the method described earlier in "Reading Data." When the desired header is matched, the CPU starts an update operation of the data portion of the sector and, before turning on the write current, times the GAP1 area. The 5-byte data block sync area is written. Next the 10-byte data block, and then 512 bytes of sector data are written from the preset location in memory. As the data is written, the CPU also creates the 2-byte checksum, which is written at the end of the data section.

The CPU also controls the trim erase timing of the read/write head. The purpose of trim erase is to erase any remaining portion of the old data section that was recorded from the sides of the new data section. At the end of the update, the write current is turned off, and, about 31 byte times later, the trim erase is turned off.

Verification

In order to ensure reliable data storage, all sector updates are followed by a verify operation. A verify operation is similar to a read operation, except that the data in memory is compared to the read disk data being transferred to memory. If any of the bytes do not compare correctly with the data in memory, an error is flagged, and an error recovery is performed by the CPU.

Formatting

A blank or new diskette must be formatted before it can be used. (Some programs, such as DCOPY, perform the formatting function implicitly.) Formatting is done by writing control information and dummy data blocks to all 80 tracks on the disk (see the "Track Format" and "Sector Format" sections under "Physical Description"). The format is a variable number of sectors per zone in soft sectored format. In order to achieve maximum speed tolerance on each diskette, the CPU performs an adaptive format procedure. Diskette speed variation (from unit to unit) causes the number of bytes on a track to vary. During format this problem is solved by always providing a fixed number of unused bytes to allow for the worst case speed. Instead of allowing the unused bytes to be wasted, the format procedure measures the size of the first track in each zone and then adjusts the gap to the size of the sector format. This causes the physical sector size to remain constant regardless of diskette speed during format. This method allows the maximum possible tolerance to speed variation without requiring a gap at the end of the track to allow for speed variation. The technique makes better use of the unused space by distributing it and using the additional intersector time to achieve stabilization of the clock recovery circuitry.

Refer to "Speed Control" and "Motor Speed Control" for more details on speed control.

Positioning

The head positioning mechanism for each drive is a four-phase stepper motor. The disk drive interface has drivers for each stepper motor which are controlled directly by the CPU. By properly sequencing the four phases of the stepper motor, the CPU can move the head of each drive in or out. All timing and control is done in software by the CPU. To reduce power consumption, the stepper motors are energized only when the drive is active; otherwise they are turned off by the CPU. The independent stepper drivers allow the CPU to perform overlapping seeks, resulting in higher system performance.

Speed Control

In order to attain maximum data capacity, the media passes under the head at a constant linear velocity. To attain this, the rotational period is varied as the radius of the track changes. The disk rotational speed is selected by the CPU. The actual speed control is performed by a single chip computer on the disk drive interface board. The CPU communicates with the speed control processor (SCP) by an eight-bit port. On system powerup, the SCP uses a default speed table that allows the system to boot. Once the operating system software is loaded, the CPU writes a new speed table to the SCP that allows it to operate with the current 512-byte sectors. The SCP can be programmed with up to 15 different speeds.

PHYSICAL DESCRIPTION

The disk interface board contains the circuitry necessary to control both of the integrated system disk drives. This circuitry consists of drive motor speed control, read/write head positioning, data decoding and encoding, read channel electronics, and write channel electronics. The interface board receives functional control from the processor unit through a dedicated I/O bus.

Motor Speed Control

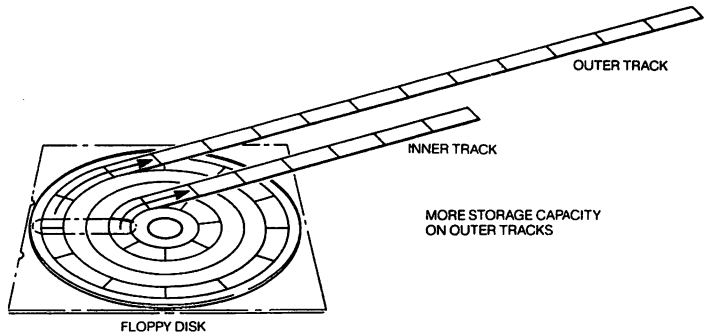
The traditional approach to storing data on floppy disks is to write data (using some encoding scheme) at a fixed rate, while rotating the disk at a constant speed. This results in several undesirable characteristics. Three major undesirable characteristics that were addressed are wasted capacity, large variation in the read signal amplitude, and low system tolerance to motor speed variation.

Since the circumference of the outermost track on the floppy is larger than the circumference of the innermost track (and, in fact, larger than all other tracks) the recording density on the outermost track is lower than on the innermost. The major limiting factor in recording on magnetic media is bit density (actually, flux reversal density), which means that the outer tracks contain less data than the inner tracks, unless adjustment is made to accommodate this problem.

Also, when the disk is rotated at a constant speed or RPM, the linear velocity of the head relative to the media varies from track to track. Since the amplitude of the recorded signal is partly a function of speed, the signal amplitude varies greatly from the outermost track (where it is highest) to the innermost track. This results in a read channel that has a lower signal-to-noise ratio than would be obtainable if all tracks were recorded with a constant amplitude signal.

These two problems are overcome by setting disk rotation speed according to the track circumference. This is done in a way that maintains a nearly constant bit density and a nearly constant linear velocity, hence a constant amplitude signal.

Figure 41: Disk Track and Sector Layout



Data written to the disk is organized into groups of 512 bytes (plus a number of synchronization and control information bytes). These groups are called sectors. Although the circumference of each track differs slightly, it is not possible to take advantage of the potential difference in capacity without using sectors of varying size. Therefore, the speed is changed only when this results in enough additional capacity for an extra sector. The disk is thus divided into groups of tracks, called zones. Each zone, when being read or written, causes the disk to rotate at a slightly different speed.

The third problem—low system tolerance to motor speed variations—is caused by a phenomena called bit shift or pulse crowding. Bit shift occurs during recording at moderately high densities. This introduces timing errors in the data transitions during subsequent reads. The clock recovery circuitry interprets these variations as motor speed error, which reduces the system's tolerance to speed variations of the drive motor.

This problem has been reduced by improving the motor speed control and using an encoding technique that is more tolerant of bit shift error. The disk rotational speed control is accomplished by using a crystal-controlled, closed-loop servo system. The servo system actually consists of two interacting closed servo loops.

The first servo loop is a fast acting inner loop, which is an analog circuit that provides excellent short-term stability. This circuit uses a charge-pump technique, which converts tach pulses from the drive motor to a voltage. This voltage is compared to a reference voltage, and any difference generates a correction in motor speed.

The second servo loop (the outer loop) digitally counts a fixed number of tach pulses from the motor, and measures the period of time that this takes. It then compares this time with the expected time. Any difference results in a modification of the reference voltage for the inner loop. This is accomplished using a single-chip microprocessor (an 8048), which uses the 5 Mhz system clock and two (8-bit) digital-to-analog converters (one per drive). Since this outer loop is crystal-referenced, it provides absolute long term stability and virtually eliminates unit to unit speed differences.

The microprocessor contains a set of speed control tables. These tables are initialized to default values at power-on and are reloadable by the processor unit.

Data Encoding Technique—GCR

To record data on magnetic media, like floppies, the data first has to be converted from the internal computer format into a form that can be stored and retrieved. This is true because data in the internal format may contain long sequences of like bits—either ones or zeroes. If data is recorded with more than a few bit times having no changes (flux reversals), the characteristics of the read channel make it impossible to read back the same signal that was recorded. Also, the data is written at a constant frequency (bit rate), but no clock signal is written. This means that the clock information must be re-created during subsequent read operations. Even though the disk speed is closely controlled (to within 2%), data transitions are required periodically to resynchronize the clock recovery circuitry.

An encoding technique called group code recording (GCR) is used to convert the data from internal representation to an acceptable form. GCR converts each (4-bit) nibble into a 5-bit code that guarantees a recording pattern that never has more than two zeroes together. Then data is recorded on the disk by causing a flux reversal for each "one" bit and no flux reversal for each "zero" bit.

Read Channel

The read channel consists of a magnetic pickup (read/write head), an amplifier section, a clock recovery section, a serial to parallel converter, and a 10-bit to 8-bit (GCR to internal form) conversion section.

The read/write head picks up a low amplitude (approximately 2 to 8 millivolts) signal from the disk. This signal is amplified differentially (to minimize the effects of common mode noise), and pass-band filtered (to reduce noise at frequencies other than those of interest). The linear output from the filter is passed to the differentiator, which generates a wave form whose zero crossovers correspond to the peaks of the read signal (these peaks occur approximately where the flux reversals take place during the write). Then this signal is fed to the comparator and digitizer circuitry. The comparator and digitizer circuitry generate a 1-microsecond read data pulse, corresponding to each peak of the read signal. These pulses serve two purposes: first, each of these pulses represents a "one" bit and so sets the serial data latch (to one); second, these pulses are used by the clock recovery circuit to keep a phase-locked loop (PLL) synchronized to the data being read from the disk. At each clock cycle (bit time), the serial data latch is shifted into the serial to parallel converter, and the serial data latch is reset (to zero).

When 10 bits have been shifted into the serial to parallel converter, the data is converted back into the original 8-bit byte. This data byte is latched, and a signal is sent to the processor unit that a byte is ready to be read.

Write Channel

The write channel consists of an 8-bit to 10-bit (internal form to GCR) code conversion section, a parallel to serial converter, write/erase current control, and the read/write head. The write circuitry is configured so that it is impossible to enable the write current if the diskette is write-protected. The write circuitry also initializes to read mode at power-up, and is prevented from writing until the power has stabilized.

Sector Format

Figure 42 illustrates sector format; Table 6 describes the parts of the sector.

Figure 42: Sector Format

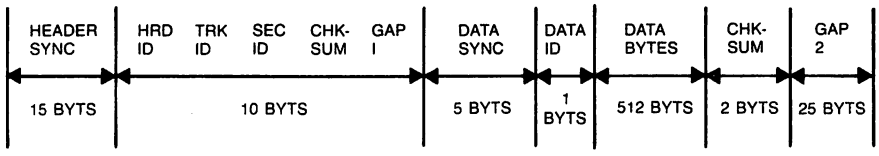


Table 6: Sector Components

COMPONENT	DESCRIPTION
Header sync	This sync mark synchronizes the PLL and causes sync detect interrupts to be sent to the CPU.
Sector header (header ID, track ID, sector ID, and checksum)	This area of 4 bytes contains sector indentification information.
Gap 1	This gap allows time for the CPU to process the sector header in formation and for the read/write head to clear the header for an update.
Data Sync	This sync mark synchronized the PLL and indicates the start of the data field.
Data field (data sync, data ID, data bytes, and checksum)	This is the useful data content of the sector for error detection if a 2-byte checksum is used.
Gap 2	This gap allows for speed variation during an update so that the next sector sync mark is not overwritten.

Track Format

Table 7 presents track format:

Table 7: Track Format

ZONE NUMBER	TRACK NUMBERS LOWER HEAD (STANDARD)	UPPER HEAD	SECTORS PER TRACK	ROTATIONAL PERIOD (MS)
0	0-3	(unused)	19	237.9
1	4-15	0-7	18	224.5
2	16-26	8-18	17	212.2
3	27-37	19-29	16	199.9
4	38-48	30-40	15	187.6
5	49-59	41-51	14	175.3
6	60-70	52-62	13	163.0
7	71-79	63-74	12	149.6
8	unused	75-79	11	144.0

Physical Bus Interface

The disk drive interface board connects to the CPU board via a 50-pin ribbon cable. This cable carries the data bus, address lines, and control signals needed to interface to the three 6522's on the interface board. All the I/O ports of the CPU System are memory-mapped, allowing more efficient I/O operations.

POWER SUPPLY

The power supply for is designed for operational and equipment safety, single-switch operation, and data protection.

The power supply is a 4 voltage regulator with one +5V output, two +12V outputs, and one -12V output. Overall feedback regulates all outputs by sensing the +5V. The -12V output and one of the +12V outputs have independent series regulators.

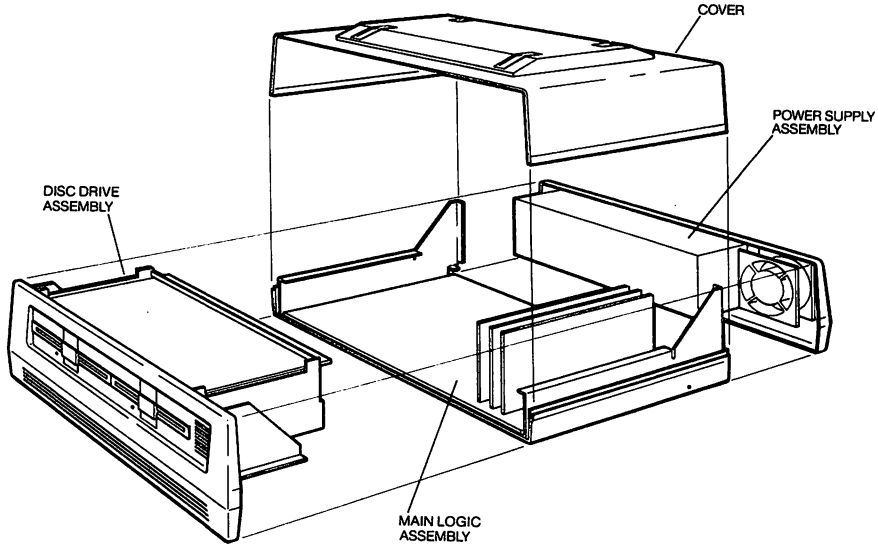
The power supply provides 6 amps of +5V +2%, 2 amps of +12V ±5%, 1.5 amps of +12V ±5%, and .2 amp of -12V ±5%. The operating range is 90-137Vac or 190-270Vac. The range may be selected and strapped by jumper wire. The power supply operates at 47-63 Hz. All power levels are regulated with overvoltage and overcurrent protection.

Line filters provide noise/ripple suppression and conducted/radiated radio frequency energy reduction.

When the power supply is shorted or overloaded, fold-back limiting occurs, preventing overheating. The unit withstands shorted output for an indefinite period and transients of up to 6000V peak. The power supply absorbs transients without causing any deviation at the output.

As shown in Figure 43, the power supply is in a shielded case, housed in the rear of the processor unit. The power supply module contains a fuse, a power switch and a line filter connector which connects to the AC power mains. It powers the processor unit, installed options, the display unit, and the keyboard unit. A 4-inch fan, mounted in the right rear of the processor unit, provides cooling air flow.

Figure 43: Processor Unit



3

DISPLAY UNIT

3. DISPLAY UNIT

The video display unit is supported by a swivel ramp and fits on top of the processor unit. The swivel ramp permits the video display unit to be swiveled right or left and to be tilted up or down. A fabric grid on the face of the CRT reduces glare and reflection and increases character contrast.

A coiled cord with a locking connector plugs the video display unit into the processor unit. The cord carries power and video signals, sync signals, and brightness control signals to the video display unit.

The video display system uses +12V power at approximately 1.2 amps. The horizontal sweep rate is approximately 15KHz. A vertical refresh rate of 76 Hz, or 76 frames per second, prevents visual flicker.

An interlace method of display is used. Each frame contains half the picture. This is very similar to what happens on a conventional television and permits a high-resolution 400-line vertical capability.

Display brightness and contrast are both software adjustable. Brightness, controlled by signals sent from the processor unit's display section, may be varied to two intensities. Contrast is controlled on the main logic board of the processor unit. The user may select eight levels of contrast from the keyboard.

4

KEYBOARD UNIT

4. KEYBOARD UNIT

The function of the keyboard is to generate and send coded electrical signals to the processor unit as each key is depressed or released. The keyboard is entirely reconfigurable.

The keyboard unit is approximately 19 inches wide, 1.8 inches high, and 6.4 inches deep. It is connected to the rear of the processor unit by a coiled cord.

The key switch is a high reliability capacitive-type switch on the keyboard. There is no mechanical contact. The signal is detected electrically, so the switch has a very long life.

Key surfaces are sculpted for comfortable typing. Key caps are removable and interchangeable, facilitating service and allowing the keyboard to be customized.

The keyboard unit is organized into five key groups. The central key group is arranged in a standard typewriter configuration. A numeric/calculator keypad is located at the far right of the keyboard. The general function keys across the top row are double-sized and can be defined for specific purposes by applications programs. A single column of specific function keys are located on the far left of the keyboard. Editing and cursor-control function keys are located in a double column between the typewriter keyboard and the numeric/calculator keypad groups.

The coiled cord is the conduit for all of the keyboard unit's inputs and outputs. The keyboard unit receives power and ground signals, a shield signal which protects the keyboard from static discharge and radiating noise, and three handshake or data control signals which control data transfer from the keyboard to the processor unit.

The comunication between the processor unit and the keyboard unit is serial. The transmission is in 9-bit words. The first eight bits are the data byte, with the least significant bit transmitted first. The last bit is a stop bit.

The keyboard returns key numbers and key status through the eight data bits. The most significant bit of the key number returned by the keyboard unit is status which flags a key "close" or a key "open." The least significant seven bits are the key number.

A single-chip microprocessor in the keyboard unit scans the keyboard for key closures and communicates with the processor unit. Keyboard status communicated to the processor unit is completely independent of key condition. The microprocessor reports an event, such as a key making or breaking contact, and the processor unit determines what that key's function is, based on application program definition.

The keyboard unit processor has an event buffer. It buffers events in case activity is going on in the processor unit that prevents it from servicing all the event signals coming in.

The communication protocol is accomplished through the use of three signal lines. The first control line passes the data serially. The second control line from the keyboard indicates to the processor unit that an event signal is ready, and the processor unit acknowledges this, using the third signal as a handshake. This return line from the processor unit to the keyboard unit is called the acknowledge line. It tells the keyboard that the processor unit has taken the bit and is making the appropriate handshake.

A protocol is defined for handling overflow problems (when the keyboard unit overflows its buffer). The protocol allows the keyboard to enter a "hold-off" state, thus permitting the processor to complete an activity without losing any event signals.

The keyboard can be made to time-out and retransmit event signals in case of an error or a problem in the handshake. The keyboard processor supports N-key rollover, which means that status is reported as the keys are depressed and as they are released. As long as the event queue doesn't overflow and the processor unit keeps up with the event queue, an unlimited number of keys can be rapidly depressed.

APPENDIXES